2000

An inquiry into the development of expert systems in legal reasoning

Trevor K. Sheeley
University of Northern Iowa

An Inquiry into the Development of Expert Systems in Legal Reasoning

Trevor K. Sheeley
Department of Computer Science, University of Northern Iowa

Computer programs have been applied to law for a number of applications, ranging from unrefined information retrieval to sophisticated legal reasoning. The latter is achieved by applying artificial intelligence. There are many appealing reasons for the choice of legal reasoning as an arena for artificial intelligence. First, the domain of law has a tendency to analyze its own reasoning process (Rissland 1988). This provides a good basis for expert systems scientists, whose first task in any domain is to discover how the experts think. Second, law is an interesting domain for artificial intelligence because legal events occupy a unique niche between phenomena that can be explained by deduction, and the happenings of less formal realms, such as common sense, which suggest few discernable patterns. Hence, coming to a better understanding of how to model the legal domain will provide a significant step toward finding patterns in phenomena heretofore evasive of human understanding (Ashley 1990). While the field of artificial intelligence benefits in these and other ways from studying law, legal professionals are the recipients of the obvious benefits provided by intelligent assistants. One day, human legal experts and their artificial assistants will work side by side to interpret legal situations.

The first goal of this paper is to review some of the steps necessary in developing a system that reasons effectively in some domain of law. The paper will begin by addressing the issues of domain selection, domain analysis and knowledge acquisition, knowledge representation, and selection of an inference method. After presenting a brief

argument against using rule-based reasoning as the primary mode of inference, the paper will go on to expound the virtues of Kevin Ashley's HYPO, a software model of case-based legal argument. It will conclude with a short description of my experience implementing part of a software model of legal reasoning.

I. Several factors must be considered when choosing a legal domain to model with an expert system. Susskind (1987) suggests a few guidelines, discussed in the paragraphs to follow.

First, the sources from which knowledge of the domain can be gathered should be limited and easily located. This is not to say that knowledge acquisition is ever easy, it simply suggests that a smaller volume of information is easier to review. Second, the domain should be one whose questions are frequently difficult enough to warrant a call upon artificial intelligence to help suggest answers. If there exists a legal domain where the predicates are definitive and the rules dictating the relationships between those predicates are fixed, a simple deduction method is all that is required. With this in mind, it might appear that statutory domains are not appropriate areas for artificial intelligence. The truth is that the interpretation of statutes does present many interesting questions (Dias, 1979; Gardner, 1987; Skalak and Rissland, 1992), such as how to resolve the open-textured predicates that inevitably complicate otherwise easy decisions in even the most formalized of legal domains.

Third, a small domain should be covered in detail, rather than covering a larger area on a superficial level. In other words, the system should indeed be an expert system, characterized by depth of knowledge and reasoning skills, not general understanding.

Fourth, the scope of the domain should be well-defined. Modeling a domain with firm boundaries increases the system's validity because the scope of the knowledge represented will not be subject to disagreement among human experts. Furthermore, it simplifies the knowledge acquisition process.

Susskind adds, as part of his fourth principle, that a legal domain free of conflicting rules is preferred over one in which similar facts tend to result in diverging legal consequences. Gardner (1987) chose to model 'offer and acceptance' problems in part because contract law is well-established. Contract law is consistent to the extent that it has been reviewed and synthesized twice this century in the Restatement of Contracts. McCarty, the creator of TAXMAN, which deals with tax law as it applies to a specific class of corporate transaction, agrees that domains composed mainly of formal concepts lend themselves best to expert systems (Sergot, 1991).

While it is important that the chosen domain's rules do not directly contradict themselves, it is equally important to select a domain in which there can be competing interpretations of a case's facts. As Susskind says himself in principle number two, the selected domain should contain problems complex enough to warrant the use of artificial intelligence techniques. An expert system should preserve the reality that legal experts are expected to disagree on hard questions (Gardner, 1987).

Besides the criteria Susskind mentions, there are additional factors to consider when choosing an appropriate domain. One such consideration is the degree to which certainty of the facts presented in the domains' lawsuits can be determined (Gardner, 1987). For example, the evidence in a contracts case is often as concrete as a legal

document in black-and-white, while the facts composing an alleged killer's alibi are only as dependable as the testimony of biased witnesses.

II. After choosing an area of law to study, the next step involves knowledge acquisition through a thorough analysis of the domain. The goal is to gather the knowledge that the system needs to preserve the integrity of domain's methods of inference.

Acquisition of knowledge about the law is facilitated by the domain's tendency to publish, codify, and index its statutes and proceedings. These can be located in encyclopedias, treatises, case books, and computerized databases. If the chosen domain is statutory, the rules of the domain may be clearly stated. While many domains have no formal rules, some have attempted to reduce the implications of the body of precedents to specific principles (Rissland, 1988). Books containing such syntheses, such as the *Restatement of Contracts* and the *Restatement of Torts,* are essential to the artificial intelligence researcher interested in the domain to which they apply.

Whether or not statutes or principles are available, it is necessary for the researcher to study the precedents of the domain. One promising resource is the *Annotated Law Reports,* which contains "notes" on case topics, outlining the issues and factors relevant to the decisions (Hafner, 1987). These summaries emphasize the very information essential to the argument process that the program wishes to undertake. An example of a computerized resource for gathering case information is *Shephard's Citations.* The on-line retrieval system maintains a network of cases with links both to cases they cited and those that cited them.

III. After conducting a complete domain analysis, the next step is to decide how the acquired knowledge will be represented in the program. Research indicates that an ideal way to represent a legal knowledge would be to have build structured collection that supports conceptual retrieval of information (Hafner, 1987; Sergot 1991). The ability to retrieve information conceptually means that queries are executed based on the meaning of the keyword, rather than mere syntactic pattern matching. The implication is that the query returns not only exact references to the keyword, but also information about its synonyms. The system's ability to understand is a result of the structure of the knowledge base.

One method for structuring a knowledge base to facilitate conceptual retrieval involves the use of frames. Beyond storing analogies, frames also represent 'has-a' relationships between objects. Many researchers have used frames to represent domain knowledge in their legal reasoning systems. In his Taxman I project, McCarty used a limited type of frame to implement tax information. The frames were "bundles of assertions" which represented only one interpretation of a given concept (Sergot, 1991). In Taxman II, McCarty extended the representation so that analogies could be drawn between related concepts (Sartor, 1993). This showed that different sets of assertions could satisfy the same predicate.

HYPO (Ashley, 1990), which models the domain of trade secrets law, also uses frames. Each case in the case knowledge base, as well as the current fact situation (cfs), is a composition hierarchy of legal case frames. The top level frame of a particular case contains attributes such as the date and holdings of the case, as well as instances of objects like the 'Corporate-Part(ies)' involved. Each 'Corporate-Party', then, has an

instance of an 'Employee' who was involved in the trade secrets issue. An 'Interpretation Frame' is used to store higher-level information yielded by analysis of the legal case frames. HYPO does not support conceptual retrieval, as the information relevant to any given query is located by requesting it from a specific instance of an object.

Rule-based systems and case-based systems alike can use frames to represent domain knowledge. For example, HYPO employs case-based reasoning, while Taxman I and II are both primarily rule-based systems. Two systems taking a mixed-paradigm approach, CABARET (Skalak and Rissland, 1992) and Gardner's (1987) program, also utilize frames to represent knowledge.

V. Many attempts have been made to model legal argument. Of the many artificial intelligence inference mechanisms used, rule-based reasoning and case-based reasoning are the most common. When the nature of legal argumentation is considered, it is evident that case-based reasoning is the better of the two choices.

Legal argument is based on the doctrine of the precedent. That is, an attorney can make arguments for a desired outcome of a case by successfully comparing the case to on-point precedents decided in the same way. Some domains of law, though, have statutes which also factor into legal decisions. With this in mind, some researchers attempting to simulate argument in statutory domains have used backward chaining as their primary inference mechanism. The following illustrations will show the limitations of the rule-based approach and the benefits of interpreting legal situations with cases.

McCarty's (1977) Taxman I project was an experiment which featured the attempt to model a particular decision in the statutory domain of U.S. corporate tax law. The inference engine is a set of rules to determine whether a transaction should be classified

as a tax-free reorganization. It makes decisions by deduction (Sergot, 1991). McCarty's original assumption was that, in a statutory domain with such well-defined predicates, there might be a single legal interpretation of a case's facts. What he discovered was that some concepts are "amorphous." This means that the concepts have no definition that can be expressed in terms of necessary and sufficient conditions. As mentioned in the previous section, Taxman II supplemented Taxman I's rules with examples. These examples allow the program to present alternative interpretations for predicates. The result is that "amorphous," or open-textured predicates, a reality in all legal domains, are modeled by Taxman II. Without examples, this aspect of legal reasoning could not have been addressed in the program.

Gardner's (1987) program, which operates in the domain of contract law, addresses questions of 'offer and acceptance.' The primary goal of the program was to model the process of how a legal question is determined to be hard, or subject to disagreement among legal experts.

The program uses rules translated from the *Restatement of Contracts* to decide whether states in the process of contract formation have been reached. For example, one of the possible states is 'proposal to modify,' which means that the offeree is interested, but would like to alter the provisions of the offer in some way. In order to resolve each of the antecedents of the rule testing the existence of a 'proposal to modify,' two sets of rules are called upon. One set of rules represents common-sense opinions about the composition of the predicates. The other is a group of examples containing both sets of conditions that represent alternative positive instances of the predicates and sets of conditions that comprise negative instances. First, the program fires any applicable

common-sense rule. Whether or not the predicate test is positive based on some common-sense rule, the program checks for examples that fire. If no examples apply, the predicate is resolved by the common-sense rule. If examples fire, and all fired examples resolve the predicate the same way, it is resolved as such. If fired examples suggest competing resolutions, the question the predicate poses is determined as unsettled by the law. This is what Gardner terms a "hard" question (Rissland 1988). Again, as in the case of Taxman II, it is examples that cause the program to acknowledge the complexity of interpreting a legal situation. Gardner is successful in modeling the way in which a legal question can be recognized as debatable. However, in her research she maintained that some questions of law are not subject to debate. Rissland points out, given a realistic number of cases, all questions would be "hard" by Gardner's definition. Gardner (1989) later acknowledged that she had underestimated the speed at which an emerging body of precedents puts a different spin on the interpretation of legal situations.

In the above two scenarios, both Gardner and McCarty arrived at the conclusion that rules have limited usefulness in an expert legal reasoning system. Outcomes of cases, even in statutory domains, do not depend on written rules. Instead, the decisions depend on intangible rules that are not formed until the current case is evaluated in light of how relevant past cases have been interpreted. Even in the unlikely event that a system developer is able to capture the implications of all the precedents in a set of written rules, it is possible that each and every new case will require the production of more rules. With these limitations of rules in mind, the researcher should consider letting the cases stand alone, each claiming authority in legal decision-making only to the extent that it is similar to the current situation. Ashley's HYPO uses case-based reasoning to

effectively model legal argumentation. The following section will summarize the benefits of a case-based approach.

VI. In order to encode cases in HYPO's case knowledge base, the first step was to decide what case information to include. HYPO is supplied with the same information contained in a squib, or case brief (Ashley, 1990). This is recorded in the legal case frames discussed in Section III above.

Beyond representing the relevant facts of each case, an expert legal reasoner needs a way to determine what those facts mean. HYPO has a mechanism to create from the underlying facts an 'Interpretation Frame' containing predicates and their values. These predicates tend not to be open-textured. For instance, an example of a predicate in the interpretation frame is 'There is a corporate defendant: Corporate-Party: Kawneer.' This value is not a product of interpretation since 'Kawneer, ' the name of a corporation, was entered by the user in the slot marked 'Corporate-Party."

Once the facts have been interpreted as higher level predicates, HYPO's thirteen top-level factors, called Dimensions, are tested for existence against the current fact situation's Interpretation Frame. For example, the Dimension 'Secrets-Voluntarily-Disclosed' only exists in the cfs if the interpretation frame indicates the satisfaction of each of its prerequisites, which include: 'There is a corporate plaintiff', 'There is a corporate defendant', 'Plaintiff makes a product', 'Plaintiff and defendant compete', 'Plaintiff has product information', and 'Plaintiff made some disclosures to outsiders.' As was the case with HYPO's resolution of predicates, no interpretation of meaning is involved in deriving Dimensions from the predicates. What this means is that the discovery of a Dimension in the cfs is a credible deductive observation that relies only on

the assumption that the information entered by the user into the legal case frames is accurate.

HYPO's Dimensions differ from the top-level factors of other systems in several ways. The Dimensions have what are called 'focal slot' prerequisites. In the above example, the focal slot prerequisite is 'Plaintiff made some disclosures to outsiders.' The focal slot prerequisite is the predicate that all of the Dimension's other prerequisites lead to. Without the other five predicates, 'Plaintiff made some disclosures to outsiders' either could not be true or would not matter. For example, if 'There is a corporate plaintiff' is not true, then there exists no corporation to make disclosures. If it is not the case that 'Plaintiff and defendant compete,' then there would be no lawsuit at all. The point is that the meaning of a Dimension lies primarily in the focal slot. Focal slots, unlike the slots for other prerequisites, may contain values other than Boolean values and unknowns. These values very often are substantive numeric values that allow for meaningful comparisons between two cases with a common Dimension.

HYPO's Dimensions differ from the top-level factors of other systems in another way. After isolating the Dimensions present in a cfs, HYPO stops using deduction. Rule-based systems designers tend to try to deduce an answer to the legal situation by writing lots of rules specifying factor values as necessary, sufficient, or prohibitive to achievement of a certain outcome of the case. They may even go so far as to include a wide range of values for the factors and strict boundaries for those values to meet the rules' conditions. Since legal argument is case-based and adversarial, a rule-based approach in which factor weights are fixed does not accurately model the law.

In HYPO, no combination of existing or absent Dimensions, or magnitudes thereof, solidifies anything about the decision on a claim. A Dimension is just a factor that has been observed in domain analysis to have some weight in a decision on the claim. A Dimension's weight is dynamic. It can only be evaluated in the context of what particular instances and/or magnitudes of that Dimension have meant to the conclusions of relevant precedents. Therefore, the Dimensions that need to be present or absent to successfully argue for the desired disposition on a claim are not apparent until the available precedents are evaluated. Moreover, the best argument that can be made depends not only on the available precedents, but also the particular arguments about the precedents made by the opposition (Ashley, 1990; Skalak and Rissland, 1992).

HYPO features a simple method for selecting the precedents that are 'most on-point,' or most similar to the cfs in terms of shared Dimensions. In order to be 'most on-point,' a precedent does not necessarily have the highest number of Dimensions in common with the cfs. 'Most on-point' simply means that the precedent has a subset of the current fact situation's Dimensions which is not a proper subset of the Dimensions of any other precedent.

An important feature of HYPO, as its name suggests, is its ability to generate hypothetical modifications of the cfs like those that attorneys employ in arguments. HYPO develops hypotheticals by characterizing precedents as near-miss along a Dimension if all the Dimension's prerequisites are satisfied except the focal slot prerequisites. Precedents are 'potentially most on-point' if they exhibit applicable Dimensions which form a maximum unique subset of the union of the current fact situation's applicable and near-miss Dimensions. With potentially most on-point

precedents, HYPO can suggest what the implications for the case would be if the modification to the current case were actual and not hypothetical. This practice of making a controlled change to the facts of the case allows the arguer to isolate the effects of that small change in light of the altered set of case facts and the body of precedents. Seeing such possibilities can notify HYPO or an attorney of alternative avenues to take in arguing the case.

In order to make the initial argument for Side 1 in a run of HYPO, any available 'best case to cite' for Side 1 is retrieved. The definition of a 'best case' is one that is most on-point and was decided in the same direction that Side 1 is advocating in the current case. This initial point for Side 1 consists of a legal conclusion regarding the fact situation, the citation of a precedent (the 'best case,' if one exists) justifying the conclusion, and an analogy between the cfs and the precedent to support the justification.

Following this point for Side 1, HYPO generates an argument for Side 2 which consists of either a counterexample or a critique of Side 1's analogy. A counterexample may do one of four things for Side 2: It may 1) trump Side 1's example by citing a case that is more on-point and was decided in Side 2's favor; 2) suggest a Hypothetical with potential to trump Side 1's example; 3) cite a precedent that shares all or some Dimensions with Side 1's example, but was decided in Side 2's favor; or 4) cite a case decided in Side 2's favor that is more extreme than Side 1's example along some shared Dimension. If HYPO does not cite a counterexample on behalf of Side 2, it distinguishes Side 1's citation from the current fact situation, calling Side 1's analogy into question. Distinguishing is done, for example, by citing a case decided in Side 2's favor that is more extreme than Side 1's example along a Dimension common to the cfs, Side 1's

citation, and itself. HYPO concludes the '3-ply argument' by stating a rebuttal on behalf of Side 1. A rebuttal is a response to any counterexamples posed by Side 2.

HYPO, by employing case-based reasoning techniques, support many capabilities desired in an expert legal reasoning system. With cases and Dimensions, HYPO is able to assign meaningful magnitudes to factors, dynamically weight the factors in the context of the cfs and its relevant precedents, present adversarial interpretations of the implications of a case, compose hypothetical modifications of the cfs to expose new lines of argument, and suggest which side in a dispute can make the best case for a decision in its favor. Not only is HYPO among the most functional of existing legal reasoning systems, but it is also very simple. While HYPO emerges as the best model for argumentation in a case-law domain, CABARET (Skalak and Rissland, 1992), which incorporates a HYPO-like system as its case-based reasoning component, appears to be a useful tool for modeling a statutory domain. However, I haven't had a chance to do any careful reading of the work of Skalak and Rissland.

VII. To learn anything concrete about the implementation of expert systems in legal reasoning, I thought it would be beneficial to actually write some code. Following the design of HYPO, I implemented over the course of the Spring 2000 semester, a simple point – counterpoint arguer in the domain of trade secrets law. My first inclination was to provide HYPO with knowledge (a precedent base and relevant factors upon which to compare the precedents) about a domain other than that of trade secrets, which Ashley used as the example in his book. It soon became apparent, however, that the time to do the domain analysis and information gathering in a domain of my choosing would require more time than I had available. With that in mind, I began corresponding with Ashley

and one of his graduate assistants at the University of Pittsburgh. After a considerable amount of hassle, Ashley finally agreed to provide me with data summarizing about 10 of his trade secrets cases. It was ironic, or perhaps fitting, that Ashley required me to sign a nondisclosure agreement regarding the data he provided, suggesting that his and my relationship could evolve into a trade secrets case in its own right. Appended to this paper is an abbreviated example of the way that my program orders precedents in terms of their relevance to the current situation, thereby generating appropriate arguments.

References

Ashley, Kevin D. 1990. *Modeling Legal Argument: Reasoning with Cases and Hypotheticals*. Cambridge: MIT Press, Bradford Books.

Ashley, Kevin D. and Edwina L. Rissland. 1988. "A Case-Based Approach to Modeling Legal Argument." *IEEE Expert*. Vol. 3, No. 3.

Bing, Jon. 1992. Book Review of *Modeling Legal Argument* by Ashley. *Artificial Intelligence and Law*. Vol. 1, No. 1.

Gardner, Anne v.d.L 1987. *An Artificial Intelligence Approach to Legal Reasoning*. Cambridge: MIT Press, Bradford Books.

Gardner, Anne v.d.L. 1989. *In the Proceedings of the Second International Conference on Artificial Intelligence and Law*. ACM Press.

Hafner, Carole D. 1987. *In the Proceedings of the First International Conference on Artificial Intelligence and Law*. ACM Press.

Johnson, Kathy A. 1988. "The Use of Case-Based Reasoning in Medical Diagnosis."

Riesbeck, Christopher K., and Roger C. Schank. 1989. "Case-Based Reasoning: An Overview." *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates.

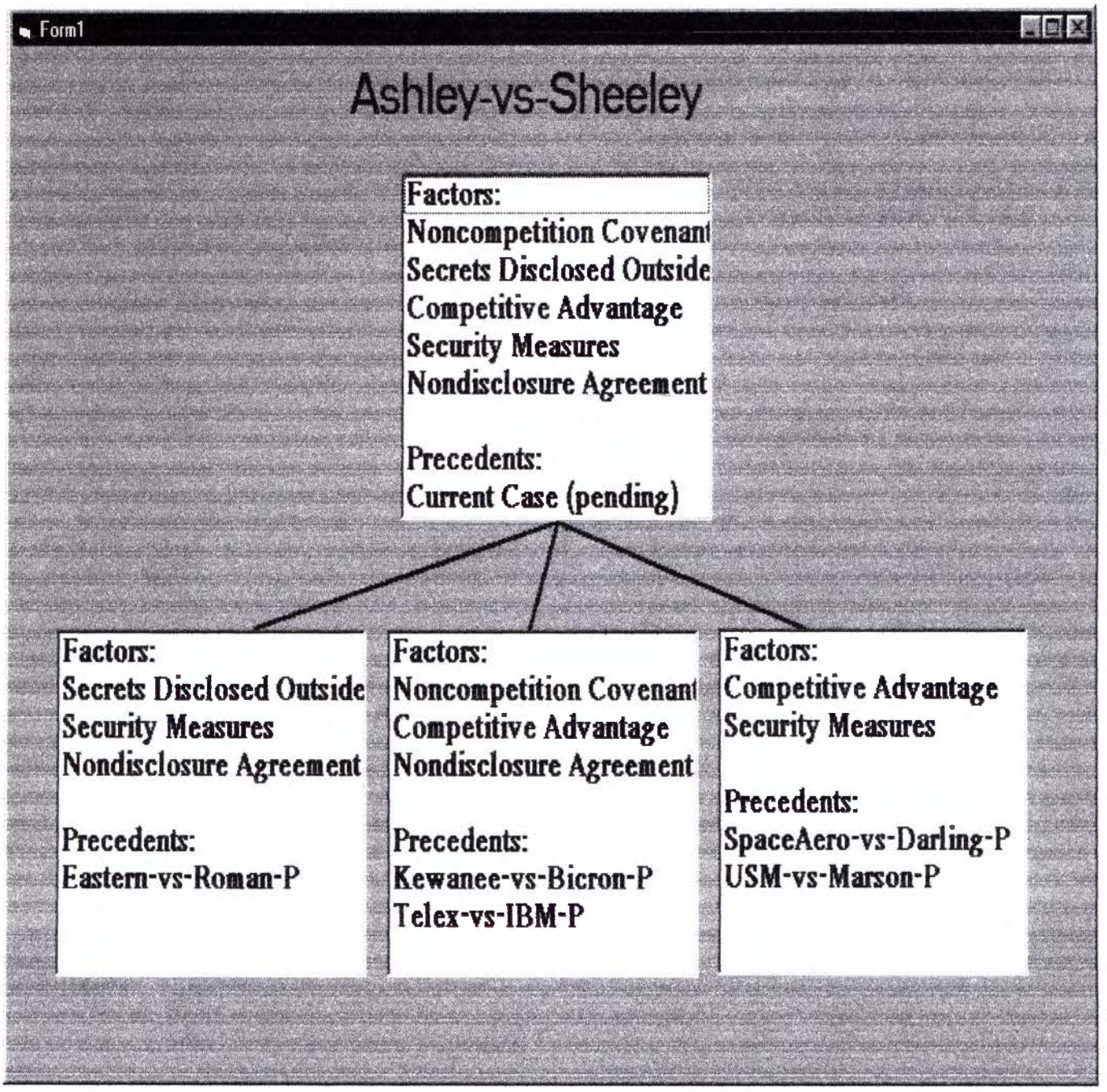
Rissland, Edwina L. 1988. "Artificial Intelligence and Legal Reasoning." *AI Magazine*. Vol. 9, No. 3: p. 45-55.

Sartor, Giovanni. 1993. *Artificial Intelligence and Law: Legal Philosophy and Legal Theory*. Tano.

Sergot, Marek. 1991. *The Representation of Law in Computer Programs.* Tano A.S.

Susskind, Richard. 1987. *Expert Systems in Law.* Clarendon Presss: Oxford.

Skalak, David B. and Edwina L. Rissland. 1992 "Arguments and Cases: An Inevitable Intertwining." *Artificial Intelligence and Law.* Vol. 1, No. 1.


Form1
Ashley-vs-Sheeley
Factors:
Noncompetition Covenant
Secrets Disclosed Outside
Competitive Advantage
Security Measures
Nondisclosure Agreement
Precedents:
Current Case (pending)
Factors:
Secrets Disclosed Outside
Security Measures
Nondisclosure Agreement
Precedents:
Eastern-vs-Roman-P
Factors:
Noncompetition Covenant
Competitive Advantage
Nondisclosure Agreement
Precedents:
Kewanee-vs-Bicron-P
Telex-vs-IBM-P
Factors:
Competitive Advantage
Security Measures
Precedents:
SpaceAero-vs-Darling-P
USM-vs-Marson-P

ASHLEY VS. SHEELEY

PLAINTIFF LEADS WITH THE POINT:

In the cfs, Ashley-vs-Sheeley, the trade secrets claim should be decided for the plaintiff because just like in Eastern-vs-Roman, which was won by the plaintiff, the PLAINTIFF ADOPTED SECURITY MEASURES TO PROTECT ITS SECRETS, WHICH HELPS THE PLAINTIFF'S CASE, and EMPLOYEE ENTERED INTO A NONDISCLOSURE AGREEMENT WITH THE PLAINTIFF. Like in Eastern-vs-Roman, the cfs should be decided for the plaintiff despite the fact(s) that THE PLAINTIFF DISCLOSED SECRETS TO OTHER PARTIES.

DEFENDANT's RESPONSE:

THE CITED CASE IS DISTINGUISHABLE FROM THE CURRENT SITUATION:

In the current fact situation, the plaintiff voluntarily disclosed their secrets to 1000 outsiders, whereas in the cited case, Eastern-vs-Roman, the plaintiff voluntarily disclosed their secrets to 100 outsiders. In the cited case, Eastern-vs-Roman, the plaintiff adopted 3 security measures, [minimal-measures, employee-nondisclosure-agreements, restrictions-on-entry-by-visitors], whereas in the current fact situation the plaintiff adopted 1 security measures, [minimal-measures].

```
import java.util.*;
public class CompetitiveAdvantage extends Dimension
{
   private String name = "Competitive Advantage";
   private String favors = "plaintiff";
   private String favordescription = "DEFENDANT UNFAIRLY SAVED DEVELOPMENT COST OR TIME
   private String description = "DEFENDANT SAVED DEVELOPMENT COST OR TIME IN DEVELOPING
   private DefendantSavedMoney defendantSavedMoney = new DefendantSavedMoney();
   private DefendantSavedTime defendantSavedTime = new DefendantSavedTime();

   public CompetitiveAdvantage()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }

   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return favordescription;
   }
   public String description()
   {
      return description;
   }

   private Vector findSubDimensions(Case thecase)
   {
      Vector subDimensions = new Vector();
      if (defendantSavedMoney.test(thecase).equals("applicable"))
         subDimensions.addElement(defendantSavedMoney);
      if (defendantSavedTime.test(thecase).equals("applicable"))
         subDimensions.addElement(defendantSavedTime);
      return subDimensions;
   }

   public String magnitude(Case thecase)
   {
      String mag = new String();
      if (findSubDimensions(thecase).contains(defendantSavedMoney))
         mag = mag + "   " + defendantSavedMoney.magnitude(thecase);
      if (findSubDimensions(thecase).contains(defendantSavedTime))
         mag = mag + "   " + defendantSavedTime.magnitude(thecase);
      return mag;
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
```

```
            return "not applicable";
        if (thecase.plaintiffProduct() == null)
            return "not applicable";
        if (thecase.defendant() == null)
            return "not applicable";
        if (thecase.defendantProduct() == null)
            return "not applicable";
        if (thecase.productsCompete() == false)
            return "not applicable";
        if (findSubDimensions(thecase).size() == 0)
            return "near miss";
        else
            return "applicable";
    }

    private Object boundarySub(Case cited, Vector luckycases)
    {
        return "no boundary";
    }
    public String comparison (String distinguisher, Case cfs, Case citedcase)
    {
        if ((defendantSavedMoney.comparison(distinguisher, cfs, citedcase).equals("not di
            && (defendantSavedTime.comparison(distinguisher, cfs, citedcase).equals("not d
        {
            return "not distinguished";
        }
        else if (defendantSavedMoney.comparison(distinguisher, cfs, citedcase).equals("no
        {
            return defendantSavedTime.comparison(distinguisher, cfs, citedcase);
        }
        else if (defendantSavedTime.comparison(distinguisher, cfs, citedcase).equals("not
        {
            return defendantSavedMoney.comparison(distinguisher, cfs, citedcase);
        }
        else
            return defendantSavedMoney.comparison(distinguisher, cfs, citedcase)
                + "   " + defendantSavedTime.comparison(distinguisher, cfs, citedcase);
    }
}
```

```
import java.sql.*;
import java.util.*;
public class Mediator2
{
   private Case cfs;
   static private Vector remainingplaintiffcases = new Vector();
   static private Vector remainingdefendantcases = new Vector();
//keep track of which cases have been cited. A case cited as an initial citation
//for a 3-ply argument can then be used later in a response, but once a case has been u
//for either purpose, it cannot be used as an initial citation, because that would
//seem redundant.

   public Mediator2(Case currentcase)
   {
      cfs = currentcase;
   }
   public Mediator2()
   {
   }
   public static void main(String [] args)
   {
      Mediator2 DrRonRoberts = new Mediator2();
      DrRonRoberts.letThemArgue();
   }

   public void letThemArgue()
   {
      CaseLibrary archives = new CaseLibrary("casefile.txt");
      if (cfs == null)
         cfs = (Case)archives.cases().elementAt(4);
      System.out.println(cfs.caseName());
      Vector applicableCases = archives.retrieveAllPrecedents(cfs.factors());
      applicableCases.removeElement((Case)cfs);
      Arguer unbiasedMessenger = new Arguer(applicableCases,cfs);
      unbiasedMessenger.findmopcs();
      remainingplaintiffcases = (Vector)unbiasedMessenger.bestpl().clone();
      if (remainingplaintiffcases.size() == 0 && unbiasedMessenger.mostpl().size() > 0)
         remainingplaintiffcases.addElement(((Vector)unbiasedMessenger.mostpl().clone()
      remainingdefendantcases = (Vector)unbiasedMessenger.bestde().clone();
      if (remainingdefendantcases.size() == 0 && unbiasedMessenger.mostde().size() > 0)
         remainingdefendantcases.addElement(((Vector)unbiasedMessenger.mostde().clone()
      //System.out.println("different side heading");
      //System.out.println("ARGUMENTS WITH PLAINTIFF " + cfs.plaintiff() + " AS SIDE 1,
      int count = 0;
      while (remainingplaintiffcases.size() > 0 && count < 1)
      {
         unbiasedMessenger.make3PlyArgument("plaintiff",(Case)remainingplaintiffcases.e
         count++;
      }
      while (remainingdefendantcases.size() > 0)
      {
         unbiasedMessenger.make3PlyArgument("defendant",(Case)remainingdefendantcases.e
      }
      while (remainingplaintiffcases.size() > 0)
      {
         unbiasedMessenger.make3PlyArgument("plaintiff",(Case)remainingplaintiffcases.e
      }
```

```
    }

    static public void addUsedCase(Case usedcase, String party)
    {
       if (party.equals("plaintiff"))
          remainingplaintiffcases.removeElement(usedcase);
       else
          remainingdefendantcases.removeElement(usedcase);
    }
}
```

```
import java.util.*;
public class Arguer
{
   private Case cfs;
   private ClaimLattice lattice;
   private Vector mostpl = new Vector();//most-on-point for the plaintiff
   private Vector bestpl = new Vector();//best cases for the defendant
   private Vector mostde = new Vector();
   private Vector bestde = new Vector();

   public Arguer(Vector appcases, Case acase)
   {

      Vector applicablecases = appcases;
      cfs = acase;
      lattice = new ClaimLattice(applicablecases, cfs);
      lattice.print();
      System.out.println("end of claim lattice");
      ///((GraphNode)lattice.root()).printDescendants();
   }

   //when an argument is initiated by side 1, first look for a best case in
   //the root of the lattice.  if there's no case for side 1 there, find a
   //lattice node on the next tier where there are no cases decided for
   //opponent. This way, Side1 opponent will not have any as-on-pt-counters
   //and Side 1 will save the case that shares a node with an opponent case
   // to be used to trump that opponent case.

   public void make3PlyArgument(String onbehalfof, Case initialcase)
   {
      String side1 = onbehalfof.toUpperCase();
      String side2;
      if (side1.equals("PLAINTIFF")) side2 = "DEFENDANT";
      else side2 = "PLAINTIFF";
      Case cite = initialcase;
      System.out.println("lead tag");
      System.out.println(side1 + " LEADS WITH THE POINT:");
      Assertion assertion = new Assertion(cite,cfs,lattice,side1.toLowerCase());
      System.out.println(assertion.toString());
      System.out.println("response tag");
      System.out.println(side2 + "'s RESPONSE:");
      Response response = new Response(assertion);
      System.out.println(response.toString());
      if (!(response.responseAssertion() == null))
      {
         System.out.println("rebuttal tag");
         System.out.println(side1 + "'s REBUTTAL:");
         Response rebuttal = new Response(response.responseAssertion());
         System.out.println(rebuttal.toString());
      }
   }
   public ClaimLattice claimLattice()
   {
      return lattice;
   }
   public Vector mostpl()
   {
```

```
    return mostpl;
}

public Vector bestpl()
{
    return bestpl;
}
public Vector mostde()
{
    return mostde;
}
public Vector bestde()
{
    return bestde;
}
public Case cfs()
{
    return cfs;
}


public void findmopcs()
{
    getmopcs((GraphNode)lattice.root());
    //System.out.println("MOP for plaintiff: " + mostpl.toString());
    getBCSp();
    //System.out.println("BCS for plaintiff: " + bestpl.toString());
    //System.out.println("MOP for defendant: " + mostde.toString());
    getBCSd();
    //System.out.println("BCS for defendant: " + bestde.toString());
}
private void getBCSd()
{
    for (int i = 0; i < mostde.size(); i++)
    {
        int count = 0;
        Case precedent = (Case)mostde.elementAt(i);
        Vector common = lattice.commonDimensions(precedent.factors(), cfs.factors());
        for (int j = 0; j < common.size(); j++)
        {
            Dimension dim = (Dimension)common.elementAt(j);
            if (dim.favors().equals("defendant"))
                count++;
        }
        if (count > 0)
            bestde.addElement((Case)mostde.elementAt(i));
    }
}

private void getBCSp()
{
    for (int i = 0; i < mostpl.size(); i++)
    {
        int count = 0;
        Case precedent = (Case)mostpl.elementAt(i);
        Vector common = lattice.commonDimensions(precedent.factors(), cfs.factors());
        for (int j = 0; j < common.size(); j++)
```

```
            {
               Dimension dim = (Dimension)common.elementAt(j);
               if (dim.favors().equals("plaintiff"))
                  count++;
            }
            if (count > 0)
               bestpl.addElement((Case)mostpl.elementAt(i));
         }
      }

      // this method sets the most-on-point cases for plaint and def
      public void getmopcs(GraphNode root)
      {
         Vector mopcases = (Vector)root.cases().clone(); // this statement and the nested
                                          // create a vector of the cases in the root, and
                                          // at level one of the claim lattice. the cases i
                                          // vector are the most on-pt cases.
         for (int k = 0; k < root.children().size(); k++)
         {
            for (int j = 0; j < ((GraphNode)root.children().elementAt(k)).cases().size();
            {
               mopcases.addElement(((GraphNode)root.children().elementAt(k)).cases().eleme
            }
         }

         // the following code separates the most on-point into those that favor the
         // plaintiff and those that favor the defendant.

         for (int i = 0; i < mopcases.size(); i++)
         {
            if (((Case)mopcases.elementAt(i)).winner().equals("defendant"))
            {
               if (!(mostde.contains((Case)mopcases.elementAt(i))))
                  mostde.addElement((Case)mopcases.elementAt(i));
            }
            else if (!(mostpl.contains((Case)mopcases.elementAt(i))))
               mostpl.addElement((Case)mopcases.elementAt(i));

         }
      }
   }
```

```
import java.util.*;
public class Assertion
{
   private Case citation;
   private Case cfs;
   private String arguingparty;
   private String opponentparty;
   private ClaimLattice originallattice;
   private GraphNode citenode;
   private Vector trumpingcounterexamples = new Vector();
   private Vector asonptcounterexamples = new Vector();
   private Vector helpers = new Vector();   //common //citation outcome good because of
   private Vector neutrals = new Vector();  //common //citation outcome good despite


   public Assertion(Case bestcase, Case cur, ClaimLattice original, String party)
   {
      Mediator2.addUsedCase(bestcase, party);
      originallattice = original;
      citation = bestcase;
      citenode = originallattice.caseNode(citation);
      cfs = cur;
      arguingparty = party;
      if (arguingparty.equals("plaintiff"))
         opponentparty = "defendant";
      else opponentparty = "plaintiff";

      sortDimensions();
      findTrumpingCounterexamples();
      findAsOnPtCounterexamples();
   }

   public Case citedCase()
   {
      return citation;
   }
   public Case cfs()
   {
      return cfs;
   }
   public String opponentParty()
   {
      return opponentparty;
   }
   public ClaimLattice claimLattice()
   {
      return originallattice;
   }
   public Vector trumpingCounterexamples()
   {
      return trumpingcounterexamples;
   }
   public Vector asOnPtCounterexamples()
   {
      return asonptcounterexamples;
   }
   public String boundaryCounterexamples()
```

```
{
   String boundarycounterexamples = new String();
   for (int i = 0; i < citation.factors().size(); i++)
   {
      Dimension dim = (Dimension)citation.factors().elementAt(i);
      if ((!(dim.boundary(citation).equals("no boundary")))
            && cfs.factors().contains(dim)) && dim.favors().equals(arguingparty))

         boundarycounterexamples = boundarycounterexamples +
            "  In the case cited by the " + arguingparty
            + ", " + dim.magnitude(citation) + ", and the " + arguingparty
            + " won.  But in " + ((Case)dim.boundary(citation)).toString()
            + ", the situation was worse for the " + opponentparty + ", since "
            + dim.magnitude((Case)dim.boundary(citation)) + ", but"
            + " the " + opponentparty + " still won the case.";
   }
   if (boundarycounterexamples.equals(""))
      boundarycounterexamples = "There are no boundary counterexamples.";
   return boundarycounterexamples;
}
public String toString() //useful explanation for initial argument (first ply)
{
   String thestring = new String();
   if (helpers.size() > 0) thestring = thestring +
         "In the cfs, " + cfs.caseName() + ", the trade secrets claim should be deci
         + " for the " + arguingparty + " because just like in " + citation.caseName
         + ", which was won by the " + arguingparty + ", " + helperDimensions() + "
   if (neutrals.size() > 0) thestring = thestring +
         "Like in " + citation.caseName() + ", the cfs should be "
         + "decided for the " + arguingparty + " despite the fact(s) that " + neutra
   return thestring;
}

private void sortDimensions()
{
   for (int i = 0; i < citation.factors().size(); i++)
   {
      Dimension dim = (Dimension)citation.factors().elementAt(i);
      if (cfs.factors().contains(dim))
      {
         if (dim.favors().equals(arguingparty))
            helpers.addElement(dim);
         else neutrals.addElement(dim);
      }
   }
}
private String helperDimensions()
{
   String help = new String();
   for (int i = 0; i < helpers.size(); i++)
   {
      Dimension dim = (Dimension)helpers.elementAt(i);
      if (i < helpers.size() - 1)
         help = help + dim.favorDescription() + ", ";
      else
         if (helpers.size() != 1)
            help = help + "and " + dim.favorDescription() + ".";
```

```
            else help = help + dim.favorDescription() + ".";
        }
        return help;
    }
    private String neutralDimensions()
    {
        String neut = new String();
        for (int i = 0; i < neutrals.size(); i++)
        {
            Dimension dim = (Dimension)neutrals.elementAt(i);
            if (i < neutrals.size() - 1)
                neut = neut + dim.description() + ",";
            else
                if (neutrals.size() != 1)
                    neut = neut + " and " + dim.description() + ".";
                else neut = neut + dim.description() + ".";
        }
        return neut;
    }

    private void findAsOnPtCounterexamples()
    {
        for (int i = 0; i < citenode.cases().size(); i++)
        {
            Case asonptcase = (Case)citenode.cases().elementAt(i);
            if (asonptcase.winner().equals(opponentparty))
            {
                asonptcounterexamples.addElement(asonptcase);
            }
        }
    }

    private void findTrumpingCounterexamples()
    {
        Vector casesfortcl = new Vector();
        casesfortcl.addElement(citation);
        Vector trumpingnodes = citenode.ancestors();
        for (int i = 0; i < trumpingnodes.size(); i++)
        {
            GraphNode trumpingnode = (GraphNode)trumpingnodes.elementAt(i);
            Vector trumpingcases = trumpingnode.cases();
            for (int j = 0; j < trumpingcases.size(); j++)
            {
                Case trumpingcase = (Case)trumpingcases.elementAt(j);
                if (trumpingcase.winner().equals(opponentparty))
                    casesfortcl.addElement(trumpingcase);
            }
        }
        ClaimLattice trumplattice = new ClaimLattice(casesfortcl, cfs);
        trumpingcounterexamples = new Vector();
        GraphNode newcitednode = trumplattice.caseNode(citation);
        //int citednodeindex = (int)trumplattice.caseNode(citation).nodenumber();
        GraphNode root = trumplattice.root();
        if (newcitednode.ancestorOf(root)) //this function name is misleading
        {
            for (int j = 0; j < root.cases().size(); j++)
            {
```

```
                Case trumptry = (Case)root.cases().elementAt(j);
                if (trumptry.winner().equals(opponentparty))
                    trumpingcounterexamples.addElement(trumptry);
            }
        }
    }
}
```

```
import java.util.*;
public class Response
{
   private String respondingparty;
   private String opponentparty;
   private Case countercase;           //counterexample
   private Case citedcase;
   private String stringDistinctions = new String();
   private Case cfs;
   private Assertion opponentassertion;//opponent's case citation
   private Assertion reply;            //citation of a counterexample

   public Response(Assertion theirassertion)
   {
      opponentassertion = theirassertion;
      respondingparty = opponentassertion.opponentParty();
      if (respondingparty.equals("plaintiff")) opponentparty = "defendant";
      else opponentparty = "plaintiff";
      citedcase = opponentassertion.citedCase();
      cfs = opponentassertion.cfs();
      if ((opponentassertion.trumpingCounterexamples().size() > 0)
         || (opponentassertion.asOnPtCounterexamples().size() > 0))
      {
         if (opponentassertion.trumpingCounterexamples().size() > 0)
         {
            countercase = (Case)opponentassertion.trumpingCounterexamples().firstElemen
         }
         else
         {//determine a mechanism for choosing the best as on pt counterexample!!
            countercase = (Case)opponentassertion.asOnPtCounterexamples().firstElement(
         }
         reply = new Assertion(countercase,cfs,opponentassertion.claimLattice(),respond
      }
      discoverDistinctions();
   }

   public Assertion responseAssertion()
   {
      return reply;
   }
   private void discoverDistinctions()
   {
      Vector distinctions = new Vector();
      // the following statement and for loop gather the union of the dimension set
      // existing in the cfs and the set existing in the case cited by opponent
      Vector considereddimensions = (Vector)cfs.factors().clone();
      for (int i = 0; i < citedcase.factors().size(); i++)
      {
         Dimension dim = (Dimension)citedcase.factors().elementAt(i);
         if (!(considereddimensions.contains(dim)))
         {
            considereddimensions.addElement(dim);
         }
      }
      for (int j = 0; j < considereddimensions.size(); j++)
      {
         Dimension focusdimension = (Dimension)considereddimensions.elementAt(j);
```

```
            if (!(focusdimension.comparison(respondingparty, cfs, citedcase).equals("not d
            {
               distinctions.addElement(focusdimension.comparison(respondingparty, cfs, cit
            }
         }
         if (!(distinctions.size() == 0))
            stringDistinctions = "WAYS IN WHICH THE CITED CASE IS "
               + "DISTINGUISHABLE FROM THE CURRENT SITUATION:  ";
         for (int k = 0; k < distinctions.size(); k++)
         {
            stringDistinctions = stringDistinctions + (String)distinctions.elementAt(k) +
         }
      }
      public String toString()
      {
         if (!(reply == null))
         {
            if (reply.claimLattice().greaterCoverage(reply.claimLattice().caseNode(counter
                              reply.claimLattice().caseNode(opponentassertion.citedCase
            {
               return "The case, " + countercase.caseName() + ", is more analogous to the
                  + "fact situation than the case just cited by our opponent.  "
                  + reply.toString()
                  + stringDistinctions
                  + "BOUNDARY COUNTEREXAMPLES:" + opponentassertion.boundaryCounterexample
            }
            else
               return "COUNTEREXAMPLE: " + reply.toString()
                  + stringDistinctions
                  + "BOUNDARY COUNTEREXAMPLES:" + opponentassertion.boundaryCounterexample
         }
         else
            return stringDistinctions
               + "BOUNDARY COUNTEREXAMPLES:" + opponentassertion.boundaryCounterexamples()
      }
   }
```

```
import java.util.*;
public class ClaimLattice
{

   private GraphNode rootNode;            // the root of the claim lattice
   private Vector applicableCases;        // potentially useful precedents to the cfs
   private Case cfs;
   private int nodenumbers = 1;
   public Vector graphNodeList = new Vector();      //nodes with their numbers as keys
   private Hashtable nodeHashtable = new Hashtable();//nodes with their cases as keys


   public ClaimLattice(Vector archives, Case currentCase)
   {
      cfs = currentCase;
      rootNode = new GraphNode(cfs.factors());
      applicableCases = archives;
      construct();
   }

   public GraphNode root()
   {
      return rootNode;
   }

   public GraphNode nodeAt(int index)
   {
      return (GraphNode)graphNodeList.elementAt(index);
   }
   public void print()
   {
      for (int i = 0; i < graphNodeList.size(); i++)
      {
         ((GraphNode)graphNodeList.elementAt(i)).print();
      }
   }

   public GraphNode caseNode(Case keycase)
   {
      return (GraphNode)nodeHashtable.get(keycase);
   }

   private void construct()
   {
      for (int c = 0; c < applicableCases.size(); c++)
      {
         Vector dim = commonDimensions(cfs.factors(),((Case)applicableCases.elementAt(c
         GraphNode newnode = new GraphNode(dim);
         newnode.addCase((Case)applicableCases.elementAt(c));
         Vector newroots = new Vector();
         GraphNode root = rootNode;
         newroots.addElement(root);
         int rootIndex = -1;
         do {
            rootIndex++;
            root = (GraphNode)newroots.elementAt(rootIndex);
            if (equalCoverage(newnode, root))
```

```
            {
               root.addCase((Case)newnode.cases().elementAt(0));
            }
            else if (lesserCoverage(newnode, root))
            {
               if (root.children().size() == 0)
                  root.addChild(newnode);
               else
               {
                  int count = 0;
                  int numChildren = root.children().size();
                  for (int i = 0; i < root.children().size(); i++)
                  {
                     if (greaterCoverage(newnode, (GraphNode)root.children().elementAt(
                     {
                        newnode.addChild((GraphNode)root.children().elementAt(i));
                        root.removeChild((GraphNode)root.children().elementAt(i));
                     }
                     else if (((lesserCoverage(newnode, (GraphNode)root.children().elem
                                 || (equalCoverage(newnode, (GraphNode)root.children().e
                              || (overlap(newnode, (GraphNode)root.children().elementAt(
                     {
                        newroots.addElement(root.children().elementAt(i));
                        if ((lesserCoverage(newnode, (GraphNode)root.children().element
                           || (equalCoverage(newnode, (GraphNode)root.children().elemen
                           count++;
                     }
                  }
                  if (count == 0)
                     root.addChild(newnode);
               }
            }
            else if (overlap(newnode, root))
            {
               for (int j = 0; j < root.children().size();j++)
               {
                  if (greaterCoverage(newnode, (GraphNode)root.children().elementAt(j))
                     newnode.addChild((GraphNode)root.children().elementAt(j));
                  else if (overlap(newnode, (GraphNode)root.children().elementAt(j)))
                     newroots.addElement(root.children().elementAt(j));
               }
            }
         }
         while (rootIndex != newroots.size()-1);
      }
      rootNode.receiveNodeNumber(nodenumbers);
      nodenumbers++;
      graphNodeList.addElement(rootNode);
      assignNodeNumbers(rootNode);
      assignKeysToNodes();
      rootNode.makeDescendantsRecognizeAncestors();
   }
   private void assignNodeNumbers(GraphNode aNode)
   {
      for (int i = 0; i < aNode.children().size(); i++)
      {
         if (!(graphNodeList.contains((GraphNode)aNode.children().elementAt(i))))
```

```
        {
            ((GraphNode)aNode.children().elementAt(i)).receiveNodeNumber(nodenumbers);
            graphNodeList.addElement((GraphNode)aNode.children().elementAt(i));
            nodenumbers++;
            assignNodeNumbers((GraphNode)aNode.children().elementAt(i));
        }
    }
}
private void assignKeysToNodes()//cases as keys to nodes
{
    for (int i = 0; i < graphNodeList.size(); i++)
    {
        Vector cases = ((GraphNode)graphNodeList.elementAt(i)).cases();
        for (int j = 0; j < cases.size(); j++)
        {
            nodeHashtable.put((Case)cases.elementAt(j),(GraphNode)graphNodeList.element
        }
    }
}

//coverage comparison methods
public boolean greaterCoverage(GraphNode newnd, GraphNode newrt)
{
    return ((this.hasAllDimensions(newnd.dimensions(),
                                commonDimensions(newrt.dimensions(), cfs.factors())))
            && (commonDimensions(newnd.dimensions(), cfs.factors())).size() >
               (commonDimensions(newrt.dimensions(), cfs.factors())).size());
}

private boolean equalCoverage(GraphNode newnd, GraphNode newrt)
{
    return ((this.hasAllDimensions(newnd.dimensions(),
                                commonDimensions(newrt.dimensions(), cfs.factors())))
            && (this.hasAllDimensions(newrt.dimensions(),
                                commonDimensions(newnd.dimensions(), cfs.factors())))
}

private boolean lesserCoverage(GraphNode newnd, GraphNode newrt)
{
    if ((this.hasAllDimensions(newrt.dimensions(),
                            commonDimensions(newnd.dimensions(),cfs.factors()))) &&
            (!(this.hasAllDimensions(newnd.dimensions(),
                            commonDimensions(newrt.dimensions(),cfs.factors())))))
        return true;
    else
        return false;
}

private boolean disjoint(GraphNode newnd, GraphNode newrt)
{
    return commonDimensions((commonDimensions(newnd.dimensions(), cfs.factors())),
                        commonDimensions(newrt.dimensions(), cfs.factors())).size()
}

private boolean overlap(GraphNode newnd, GraphNode newrt)
{
    return ((((!(greaterCoverage(newnd, newrt))) && (!(equalCoverage(newnd, newrt))))
```

```
            && ((!(lesserCoverage(newnd, newrt))) && (!(disjoint(newnd, newrt)))));
    }

    // this method checks to see if dimensions1 is a superset of dimensions2
    private boolean hasAllDimensions(Vector dimensions1, Vector dimensions2)
    {
       for (int i = 0; i < dimensions2.size(); i++)
       {
          if (!(dimensions1.contains(dimensions2.elementAt(i))))
             return false;
       }
       return true;
    }

    public Vector commonDimensions(Vector theseDimensions, Vector case2dimensions)
    // returns the dimensions shared by the case and anothercase
    {
       Vector set1 = theseDimensions;
       Vector set2 = case2dimensions;
       if (theseDimensions.size() > case2dimensions.size())
       {
          set1 = case2dimensions;
          set2 = theseDimensions;
       }
       Vector common = new Vector();
       for (int i = 0; i < set2.size(); i++)
       {
          if (set1.contains(set2.elementAt(i)))
             common.addElement(set2.elementAt(i));
       }
       return common;
    }
}
```

```
import java.util.*;
public class GraphNode
{
   private int nodenumber;
   private Vector dimensionList;
   private Vector associatedCases = new Vector();
   private Vector parents = new Vector();
   private Vector ancestors = new Vector();
   private Vector children = new Vector();

   public GraphNode(Vector dimensions)
   {
      dimensionList = dimensions;
   }

   public int nodenumber()
   {
      return nodenumber;
   }

   public Vector dimensions()
   {
      return dimensionList;
   }

   public Vector cases()
   {
      return associatedCases;
   }

   public Vector children()
   {
      return children;
   }
   public boolean ancestorOf(GraphNode node)
   {
      return ancestors.contains(node);
   }

   public Vector ancestors()
   {
      return ancestors;
   }

   public void addChild(GraphNode child)
   {
      children.addElement(child);
   }

   public void removeChild(GraphNode child)
   {
      children.removeElement(child);
   }

   public void addParent(GraphNode parent)
   {
      parents.addElement(parent);
```

```
    }

    public void addAncestor(GraphNode ancestor)
    {
       if (!(ancestors.contains(ancestor)))
          ancestors.addElement(ancestor);
    }

    public void addCase(Case acase)
    {
       if (!(associatedCases.contains(acase)))
          associatedCases.addElement(acase);
    }

    public void setDimensions(Vector dimensions)
    {
       dimensionList = dimensions;
    }

    public void receiveNodeNumber(int number)
    {
       nodenumber = number;
    }

    public void print()
    {
       System.out.println("new node");
       //System.out.println("GraphNode " + nodenumber);
       System.out.println("Factors:");
       listDimensions();
       System.out.println("");
       System.out.println("Precedents:");
       if (nodenumber == 1)
       {
          System.out.println("Current Case (pending)");
       }
       for (int j = 0; j < associatedCases.size(); j++)
       {
          String won = ((Case)associatedCases.elementAt(j)).winner();
          if (won.equals("plaintiff")) won = "-P";
          else won = "-D";
          System.out.println(((Case)associatedCases.elementAt(j)).caseName() + won);//"
                             //+ ((Case)associatedCases.elementAt(j)).winner());
       }
       System.out.println("child nodes:");
       System.out.println(children.size());

       for (int i = 0; i < children.size(); i++)
       {
          System.out.println(((GraphNode)children.elementAt(i)).nodenumber());
          //if (!(i == parents.size()-1)) System.out.print(", ");
       }
       //System.out.print("\n");
       //System.out.println("*****************************************************");
    }

    public void printDescendants()
```

```
    {
       print();
       for (int i = 0; i < children.size(); i++)
       {
          ((GraphNode)children.elementAt(i)).printDescendants();
       }
    }

    public void makeDescendantsRecognizeAncestors()
    {
       for (int i = 0; i < children.size(); i++)
       {
          for (int j = 0; j < ancestors.size(); j++)
          {
             ((GraphNode)children.elementAt(i)).addAncestor((GraphNode)ancestors.element
          }
          ((GraphNode)children.elementAt(i)).addParent(this);
          ((GraphNode)children.elementAt(i)).addAncestor(this);
          ((GraphNode)children.elementAt(i)).makeDescendantsRecognizeAncestors();
       }
    }

    public void listDimensions()
    {
       for (int i = 0; i < dimensionList.size(); i++)
       {
          System.out.println(((Dimension)dimensionList.elementAt(i)).toString());
       }
    }

}
```

```
import java.io.*;
import java.util.*;
public class CaseReader

{
   private FileReader theReader;
   private StreamTokenizer tokenizer;

   public CaseReader(String filename) throws IOException
   {
      theReader = new FileReader (filename);
      tokenizer = new StreamTokenizer(theReader);
   }

   public Case readcase() throws IOException
   {
      Case theCase = new Case();
      tokenizer.nextToken();
      if (tokenizer.sval.equals("casename"))
      {
         tokenizer.nextToken();
         theCase.setCaseName((String) tokenizer.sval);
      }
      else
      {System.out.println("bad file");}
      tokenizer.nextToken();
      tokenizer.nextToken();
      theCase.setPlaintiff((String) tokenizer.sval);
      tokenizer.nextToken();
      tokenizer.nextToken();
      theCase.setDefendant((String) tokenizer.sval);
      tokenizer.nextToken();
      tokenizer.nextToken();
      theCase.setWonBy((String) tokenizer.sval);
      tokenizer.nextToken();
      tokenizer.nextToken();
      theCase.setPlaintiffProduct((String) tokenizer.sval);
      tokenizer.nextToken();
      tokenizer.nextToken();
      theCase.setDefendantProduct((String) tokenizer.sval);
      theCase.setPartiesCompete(true);
      theCase.setProductsCompete(true);
      tokenizer.nextToken();
      tokenizer.nextToken();
      tokenizer.nextToken();
      tokenizer.nextToken();
      tokenizer.nextToken();
      tokenizer.nextToken();
      theCase.setEmployeeName((String) tokenizer.sval);
      tokenizer.nextToken();
      tokenizer.nextToken();
      if (tokenizer.sval.equals("yes"))
      {
         Vector tools = new Vector();
         tokenizer.nextToken();
         for (int t = 0; t < tokenizer.nval; t++)
         {
```

```
            tokenizer.nextToken();
            tools.addElement((String)tokenizer.sval);
         }
         theCase.setToolsTransferred(tools);
      }
      tokenizer.nextToken();
      tokenizer.nextToken();
      if (tokenizer.sval.equals("yes"))
      {
         Vector bribes = new Vector();
         tokenizer.nextToken();
         for (int b = 0; b < tokenizer.nval; b++)
         {
            tokenizer.nextToken();
            bribes.addElement((String)tokenizer.sval);
         }
         theCase.setBribery(bribes);
      }
      theCase.setDefendantAccess(true);
      tokenizer.nextToken();
      tokenizer.nextToken();
      tokenizer.nextToken();
      tokenizer.nextToken();
      theCase.setDevelopmentRole((String)tokenizer.sval);
      tokenizer.nextToken();
      tokenizer.nextToken();
      if (tokenizer.sval.equals("yes"))
         theCase.setDisclosureEvent(true);
      tokenizer.nextToken();
      tokenizer.nextToken();
      theCase.setNondisclosureAgreement((String)tokenizer.sval);
      tokenizer.nextToken();
      tokenizer.nextToken();
      if (tokenizer.sval.equals("yes"))
      {
         theCase.setDefSavedTime(true);
         tokenizer.nextToken();
         theCase.setTimeSaved((int)tokenizer.nval);
         tokenizer.nextToken();
         theCase.setPlaintTime(new Float(tokenizer.nval));
         tokenizer.nextToken();
         theCase.setDefTime(new Float(tokenizer.nval));
      }
      tokenizer.nextToken();
      tokenizer.nextToken();
      if (tokenizer.sval.equals("yes"))
      {
         theCase.setDefSavedCost(true);
         tokenizer.nextToken();
         theCase.setCostSaved((int)tokenizer.nval);
         tokenizer.nextToken();
         theCase.setPlaintCost(new Float(tokenizer.nval));
         tokenizer.nextToken();
         theCase.setDefCost(new Float(tokenizer.nval));
      }
      tokenizer.nextToken();
      tokenizer.nextToken();
```

```
        theCase.setKnowledgeType((String)tokenizer.sval);
        tokenizer.nextToken();
        tokenizer.nextToken();
        if (tokenizer.sval.equals("yes"))
        {
           Vector measures = new Vector();
           tokenizer.nextToken();
           for(int m = 0; m < tokenizer.nval; m++)
           {
              tokenizer.nextToken();
              measures.addElement((String)tokenizer.sval);
           }
           theCase.setSecurityMeasures(measures);
        }
        tokenizer.nextToken();
        tokenizer.nextToken();
        if (tokenizer.sval.equals("yes"))
        {
           theCase.setIfSecretsDisclosedOutsiders(true);
           tokenizer.nextToken();
           theCase.setNumDisclosees((int)tokenizer.nval);
        }
        tokenizer.nextToken();
        tokenizer.nextToken();
        if (tokenizer.sval.equals("yes"))
           theCase.setDisInNegotiations(true);
        tokenizer.nextToken();
        tokenizer.nextToken();
        if (tokenizer.sval.equals("yes"))
           theCase.setNoncompetitionCovenant(true);
        tokenizer.nextToken();
        tokenizer.nextToken();
        return theCase;
     }
  }
```

```
import java.io.*;
import java.util.*;
public class CaseLibrary
{
   private Vector DIMENSIONS = new Vector();
   private Vector cases;
   private String file;
   public CaseLibrary(String filename)
   {
      file = filename;
      cases = new Vector();
      getcases();
      instantiateDimensions();
      findDimensions();
   }

   public void getcases()
   // populates the library with cases from a file
   {
      try
      {
         CaseReader myCaseReader = new CaseReader(file);
         for (int i = 0; i < 12; i++)
         {
            cases.addElement((Case)myCaseReader.readcase());
         }
      }
         catch(IOException e)
         {
            System.out.println("I/O problem");
         }
   }

   public Vector cases()
   {
      return cases;
   }
   public void setCases(Vector thesecases)
   {
      cases = thesecases;
   }
   private void instantiateDimensions()
   {
      Bribery bribery = new Bribery();
      DIMENSIONS.addElement(bribery);
      NoncompetitionCovenant noncompetitionCovenant = new NoncompetitionCovenant();
      DIMENSIONS.addElement(noncompetitionCovenant);
      SecretsDisclosedOutsiders secretsDisclosedOutsiders = new SecretsDisclosedOutside
      DIMENSIONS.addElement(secretsDisclosedOutsiders);
      CompetitiveAdvantage competitiveAdvantage = new CompetitiveAdvantage();
      DIMENSIONS.addElement(competitiveAdvantage);
      SecurityMeasures securityMeasures = new SecurityMeasures();
      DIMENSIONS.addElement(securityMeasures);
      SoleDeveloper soleDeveloper = new SoleDeveloper();
      DIMENSIONS.addElement(soleDeveloper);
      NondisclosureAgreement nondisclosureAgreement = new NondisclosureAgreement();
      DIMENSIONS.addElement(nondisclosureAgreement);
```

```
        NondisclosureAgreementSpecific agreementSpecific = new NondisclosureAgreementSpec
        DIMENSIONS.addElement(agreementSpecific);
        BroughtTools broughtTools = new BroughtTools();
        DIMENSIONS.addElement(broughtTools);
    }

    private void findDimensions()
    {
        for (int i = 0; i < cases.size(); i++)
        {
            Case acase = (Case)cases.elementAt(i);
            Vector dims = new Vector();
            for (int j = 0; j < DIMENSIONS.size(); j++)
            {
                Dimension dim = (Dimension)DIMENSIONS.elementAt(j);
                if (dim.test(acase).equals("applicable"))
                {
                    dims.addElement(dim);
                    dim.addCase(acase);
                }
            }
            acase.setDimensionList(dims);
        }
    }

    public Vector retrieveAllPrecedents(Vector cfsDimensionList)
    {
        Vector precedents = new Vector();
        for (int i = 0; i < cases.size(); i++)
        {
            boolean isaprecedent = false;
            for (int j = 0; j < cfsDimensionList.size(); j++)
            {
                if(((Case)cases.elementAt(i)).hasDimension(((Dimension)cfsDimensionList.ele
                    isaprecedent = true;
            }
            if (isaprecedent) precedents.addElement(cases.elementAt(i));
        }
        return precedents;
    }
}
```

```
public class BroughtTools extends Dimension
{
   private String name = "Brought Tools";
   private String favors = "plaintiff";
   private String favordescription = "a COMMON EMPLOYEE TRANSFERRED TOOLS FROM THE PLAI
   private String description = "a COMMON EMPLOYEE TRANSFERRED TOOLS FROM THE PLAINTIFF
   private String comparisontype = "some vs none";
   private String proplaintdirection = "some";

   public BroughtTools()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return favordescription;
   }
   public String description()
   {
      return description;
   }
   public String magnitude(Case thecase)
   {
      return "The employee, " + thecase.commonEmployeeName() + ", transferred "
         + thecase.toolsTransferred().toString() + " from the plaintiff, " + thecase.pl
         + ", to the defendant, " + thecase.defendant();
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.productsCompete() == false)
         return "not applicable";
      if (thecase.commonEmployeeName() == null)
         return "not applicable";
      if (thecase.defendantAccessViaCommon() == false)
         return "not applicable";
      if (thecase.toolsTransferred().size() == 0)
         return "near miss";
      else
```

```
        return "applicable";
    }

    public String comparison (String distinguisher, Case cfs, Case citedcase)
    {
       // this comparison method is good for any dimension where the
       // comparison type is some vs. none.  The magnitude could
       // be included in the distinguishing comments too
       if (cfs.hasDimension(this) && citedcase.hasDimension(this))
       {
          return "not distinguished";
       }
       else if (cfs.hasDimension(this) && !(citedcase.hasDimension(this)))
       {
          if (favors.equals(distinguisher))
             return "In the current fact situation, " + cfs.caseName() + ", "
                + description + ".  Not so in the cited case, " + citedcase.caseName() +
          else
             return "not distinguished";
       }
       else
       {
          if (favors.equals(distinguisher))
             return "not distinguished";
          else
             return "In the cited case, " + citedcase.caseName() + ", "
                + description + ".  Not so in the current fact situation, " + cfs.caseNa
       }
    }
}
```

```
public class Bribery extends Dimension
{
   private String name = "Bribery";
   private String favors = "plaintiff";
   private String favordescription = "THE DEFENDANT BRIBED A COMMON EMPLOYEE TO LEAVE T
   private String description = "A COMMON EMPLOYEE WAS PAID TO CHANGE EMPLOYERS";
   private String comparisontype = "some vs none";
   private String proplaintdirection = "some";

   public Bribery()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return favordescription;
   }
   public String description()
   {
      return description;
   }
   public String magnitude(Case thecase)
   {
      return "The employee, " + thecase.commonEmployeeName() + ", was paid in the form
         + thecase.bribery().toString() + " to change employers, from, " + thecase.plai
         + ", to the defendant, " + thecase.defendant();
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.productsCompete() == false)
         return "not applicable";
      if (thecase.commonEmployeeName() == null)
         return "not applicable";
      //if (thecase.defendantAccessViaCommon() == false)
       //  return "not applicable";
      if (thecase.bribery().size() == 0)
         return "near miss";
      else
```

```
        return "applicable";
    }

    public String comparison (String distinguisher, Case cfs, Case citedcase)
    {
       // this comparison method is good for any dimension where the
       // comparison type is some vs. none.  The magnitude could
       // be included in the distinguishing comments too
       if (cfs.hasDimension(this) && citedcase.hasDimension(this))
       {
          return "not distinguished";
       }
       else if (cfs.hasDimension(this) && !(citedcase.hasDimension(this)))
       {
          if (favors.equals(distinguisher))
             return "In the current fact situation, " + cfs.caseName() + ", "
                + description + ".  Not so in the cited case, " + citedcase.caseName() +
          else
             return "not distinguished";
       }
       else
       {
          if (favors.equals(distinguisher))
             return "not distinguished";
          else
             return "In the cited case, " + citedcase.caseName() + ", "
                + description + ".  Not so in the current fact situation, " + cfs.caseNa
       }
    }
}
```

```
import java.util.*;
abstract class Dimension
{
   private String name = "generic";
   private String favors = "no subclass";
   private String description;
   private String favordescription;
   private String comparisontype;
   private String proplaintdirection;
   private Vector cases = new Vector();

   public Dimension()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public Vector cases()
   {
      return cases;
   }
   public void addCase(Case acase)
   {
      cases.addElement(acase);
   }

   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return favordescription;
   }
   public String description()
   {
      return description;
   }
   public String magnitude(Case thecase)
   {
      return "no concrete classes";
   }

   public String test (Case theCase)
   {
      return "don't know";
   }
   private Object boundarySub(Case cited, Vector luckycases)
   {
      return "no boundary";
   }
```

```
    // returns the case with the worst mag for this dimension for the distinguisher
    // but still decided for the distinguisher
    public Object boundary(Case cited)
    {
       Vector luckycases = new Vector();
       for (int i = 0; i < cases.size(); i++)
       {
          Case acase = (Case)cases.elementAt(i);
          if (!(acase.winner().equals(favors)))
             luckycases.addElement(acase);
       }
       return boundarySub(cited, luckycases);
    }
    public String comparison(String distinguisher, Case cfs, Case citedcase)
    {
       return "no comparison yet";
    }

}
```

```
public class DefendantSavedTime extends Dimension
{
   private String name = "Defendant Saved Time";
   private String favors = "plaintiff";
   private String description = "THE DEFENDANT UNFAIRLY SAVED DEVELOPMENT TIME";

   public DefendantSavedTime()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String description()
   {
      return description;
   }
   public String magnitude(Case thecase)
   {
      return "the defendant, " + thecase.defendant() + ", saved "
         + new Float(1- thecase.defMonthsSpent().floatValue()/thecase.plaintMonthsSpent
         + " % of the time spent by the plaintiff" ;
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.productsCompete() == false)
         return "not applicable";
      if (!(thecase.defSavedTime()))
         return "near miss";
      else
         return "applicable";
   }
   public String comparison (String distinguisher, Case cfs, Case citedcase)
   {
      if (test(cfs).equals("applicable") && test(citedcase).equals("applicable"))
      {
         float cfsDefRelativeCost = (float)(1 - cfs.defMonthsSpent().floatValue()/
                           cfs.plaintMonthsSpent().floatValue());
         float citeDefRelativeCost = (float)(1 - citedcase.defMonthsSpent().floatValue(
                           citedcase.plaintMonthsSpent().floatValue());
         if (distinguisher.equals("defendant"))
```

```
        {
            if (citeDefRelativeCost > cfsDefRelativeCost)//def saved more in citation
                return "In the case cited by the plaintiff, " + citedcase.caseName()
                    + ", " + magnitude(citedcase) + ", which is more of a savings than in
                    + "the current situation, where " + magnitude(cfs) + ".";
            else //if def saved more in the cfs
                return "not distinguished";
        }
        else
        {
            if (citeDefRelativeCost > cfsDefRelativeCost)
                return "not distinguished";
            else
                return "In the case cited by the defendant, " + citedcase.caseName()
                    + ", " + magnitude(citedcase) + ", which is less of a savings than in
                    + "the current fact situation, where " + magnitude(cfs) + ".";
        }
    }
    else if (test(cfs).equals("applicable") &&  ! (test(citedcase).equals("applicable
    {
        if (distinguisher.equals("defendant"))
            return "not distinguished";
        else
            return "In the current fact situation, " + description + ".  Not "
                + "so in the case cited by the defendant.";
    }
    else
    {
        if (distinguisher.equals("defendant"))
            return "In the case cited by the plaintiff, " + citedcase.caseName()
                + ", " + description + ".  Not so in the current fact situation.";
        else
            return "not distinguished";
    }
  }
}
```

```
public class DefendantSavedMoney extends Dimension
{
   private String name = "Defendant Saved Money";
   private String favors = "plaintiff";
   private String description = "THE DEFENDANT UNFAIRLY SAVED DEVELOPMENT COST";

   public DefendantSavedMoney()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String description()
   {
      return description;
   }
   public String magnitude(Case thecase)
   {
      return "the defendant, " + thecase.defendant() + ", saved "
         + new Float(1 - thecase.defDollarsSpent().floatValue()/thecase.plaintDollarsSp
         + " % of the dollars spent by the plaintiff" ;
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.productsCompete() == false)
         return "not applicable";
      if (!(thecase.defSavedMoney()))
         return "near miss";
      else
         return "applicable";
   }


   public String comparison (String distinguisher, Case cfs, Case citedcase)
   {
      if (test(cfs).equals("applicable") && test(citedcase).equals("applicable"))
      {
         float cfsDefRelativeCost = (float)(1 - cfs.defDollarsSpent().floatValue()/
                           cfs.plaintDollarsSpent().floatValue());
         float citeDefRelativeCost = (float)(1 - citedcase.defDollarsSpent().floatValue
```

```
                        citedcase.plaintDollarsSpent().floatValue());
        if (distinguisher.equals("defendant"))
        {
           if (citeDefRelativeCost > cfsDefRelativeCost)//def saved more in citation
              return "In the case cited by the plaintiff, " + citedcase.caseName()
                 + ", " + magnitude(citedcase) + ", which is more of a savings than in
                 + "the current situation, where " + magnitude(cfs) + ".";
           else //if def saved more in the cfs
              return "not distinguished";
        }
        else
        {
           if (citeDefRelativeCost > cfsDefRelativeCost)
              return "not distinguished";
           else
              return "In the case cited by the defendant, " + citedcase.caseName()
                 + ", " + magnitude(citedcase) + ", which is less of a savings than in
                 + "the current fact situation, where " + magnitude(cfs) + ".";
        }
     }
     else if (test(cfs).equals("applicable") &&  ! (test(citedcase).equals("applicable
     {
        if (distinguisher.equals("defendant"))
           return "not distinguished";
        else
           return "In the current fact situation, " + description + ".  Not "
              + "so in the case cited by the defendant.";
     }
     else
     {
        if (distinguisher.equals("defendant"))
           return "In the case cited by the plaintiff, " + citedcase.caseName()
              + ", " + description + ".  Not so in the current fact situation.";
        else
           return "not distinguished";
     }
  }
}
```

```
public class SoleDeveloper extends Dimension
{
   private String name = "Sole Developer";
   private String favors = "defendant";
   private String favordescription = "THE EMPLOYEE WAS THE SOLE DEVELOPER OF THE PLAINT
   private String description = "THE EMPLOYEE WAS THE SOLE DEVELOPER OF THE PLAINTIFF'S
   private String comparisontype = "some vs none";
   private String proplaintdirection = "none";

   public SoleDeveloper()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return favordescription;
   }
   public String description()
   {
      return description;
   }
   public String magnitude(Case thecase)
   {
      return "The employee, " + thecase.commonEmployeeName() + ", was the sole develope
         + "of " + thecase.plaintiffProduct();
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.productsCompete() == false)
         return "not applicable";
      if (thecase.commonEmployeeName() == null)
         return "not applicable";
      if (!(thecase.developmentRole().equals("sole-developer")))
         return "near miss";
      else
         return "applicable";
   }
```

```
    public String comparison (String distinguisher, Case cfs, Case citedcase)
    {
       // this comparison method is good for any dimension where the
       // comparison type is some vs. none.  The magnitude could
       // be included in the distinguishing comments too
       if (cfs.hasDimension(this) && citedcase.hasDimension(this))
       {
          return "not distinguished";
       }
       else if (cfs.hasDimension(this) && !(citedcase.hasDimension(this)))
       {
          if (favors.equals(distinguisher))
             return "In the current fact situation, " + cfs.caseName() + ", "
                + description + ".  Not so in the cited case, " + citedcase.caseName() +
          else
             return "not distinguished";
       }
       else
       {
          if (favors.equals(distinguisher))
             return "not distinguished";
          else
             return "In the cited case, " + citedcase.caseName() + ", "
                + description + ".  Not so in the current fact situation, " + cfs.caseNa
       }
    }
}
```

```
import java.util.*;
public class SecurityMeasures extends Dimension
{
   private String name = "Security Measures";
   private String favors = "plaintiff";
   private String favordescription = "the PLAINTIFF ADOPTED SECURITY MEASURES TO PROTEC
   private String description = "the PLAINTIFF ADOPTED SECURITY MEASURES TO PROTECT ITS
   private String comparisontype = "more vs less";
   private String proplaintdirection = "more";
   private Vector cases = new Vector();

   public SecurityMeasures()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return favordescription;
   }
   public String description()
   {
      return description;
   }
   public void addCase(Case acase)
   {
      cases.addElement(acase);
   }
   public String magnitude(Case thecase)
   {
      return "the plaintiff adopted " + thecase.securityMeasures().size() + " security
         + thecase.securityMeasures().toString();
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.productsCompete() == false)
         return "not applicable";
      if (thecase.partiesCompete() == false)
         return "not applicable";
```

```
      if (thecase.securityMeasures().size() == 0)
         return "near miss";
      else
         return "applicable";
   }
   public Object boundary(Case cited)
   {
      Vector luckycases = new Vector();
      for (int i = 0; i < cases.size(); i++)
      {
         Case acase = (Case)cases.elementAt(i);
         if (!(acase.winner().equals(favors)))
            luckycases.addElement(acase);
      }
      return boundarySub(cited, luckycases);
   }

   private Object boundarySub(Case cited, Vector luckycases)
   {
      Case boundarycase = new Case();
      int max = 0;
      for (int i = 0; i < luckycases.size(); i++)
      {
         Case acase = (Case)luckycases.elementAt(i);
         if (acase.securityMeasures().size() > max)
         {
            max = acase.securityMeasures().size();
            boundarycase = acase;
         }
      }
      if (max == 0)
         return "no boundary";
      else
      {
         if (max > cited.securityMeasures().size())
            return boundarycase;
         else
            return "no boundary";
      }
   }

//more vs less
   public String comparison (String distinguisher, Case cfs, Case citedcase)
   {
      if (cfs.hasDimension(this) && citedcase.hasDimension(this))
      {
         if (distinguisher.equals("defendant"))
         {                                        //since it favors plaintiff
            if (cfs.securityMeasures().size() < citedcase.securityMeasures().size())
            {
               return "In the cited case, " + citedcase.caseName() + ", " + magnitude(c
                  + ", whereas in the current fact situation " + magnitude(cfs);
            }
            else
               return "not distinguished";
         }
         else
```

```
        {
            if (cfs.securityMeasures().size() < citedcase.securityMeasures().size())
                return "not distinguished";
            else
            {
                return "In the current fact situation, " + magnitude(cfs)
                    + ", whereas in the cited case, " + citedcase.caseName() + ", " + mag
            }
        }
    }
    else if (cfs.hasDimension(this) && !(citedcase.hasDimension(this)))
    {
        if (favors.equals(distinguisher))
            return "In the current fact situation, " + cfs.caseName() + ", "
                + description + ".  Not so in the cited case, " + citedcase.caseName() +
        else
            return "not distinguished";
    }
    else
    {
        if (favors.equals(distinguisher))
            return "not distinguished";
        else
            return "In the cited case, " + citedcase.caseName() + ", "
                + description + ".  Not so in the current fact situation, " + cfs.caseNa
    }
  }
}
```

```
import java.util.*;
public class SecretsDisclosedOutsiders extends Dimension
{
   private String name = "Secrets Disclosed Outsiders";
   private String favors = "defendant";
   private String favordescription = "THE PLAINTIFF WAS NOT CAREFUL WITH THEIR SECRETS-
   private String description = "THE PLAINTIFF DISCLOSED SECRETS TO OTHER PARTIES";
   private String comparisontype = "more vs less";
   private String proplaintdirection = "less";
   private Vector cases = new Vector();

   public SecretsDisclosedOutsiders()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return favordescription;
   }
   public String description()
   {
      return description;
   }
   public void addCase(Case acase)
   {
      cases.addElement(acase);
   }
   public String magnitude(Case thecase)
   {
      return "the plaintiff voluntarily disclosed their secrets to " + thecase.numDiscl
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.productsCompete() == false)
         return "not applicable";
      if (thecase.partiesCompete() == false)
         return "not applicable";
      if (thecase.numDisclosees() == 0)
```

```
            return "near miss";
         else
            return "applicable";
      }
      public Object boundary(Case cited)
      {
         Vector luckycases = new Vector();
         for (int i = 0; i < cases.size(); i++)
         {
            Case acase = (Case)cases.elementAt(i);
            if (!(acase.winner().equals(favors)))
               luckycases.addElement(acase);
         }
         return boundarySub(cited, luckycases);
      }

      private Object boundarySub(Case cited, Vector luckycases)
      {
         Case boundarycase = new Case();
         int max = 0;
         for (int i = 0; i < luckycases.size(); i++)
         {
            Case acase = (Case)luckycases.elementAt(i);
            if (acase.numDisclosees() > max)
            {
               max = acase.numDisclosees();
               boundarycase = acase;
            }
         }
         if (max == 0)
            return "no boundary";
         else
         {
            if (max > cited.numDisclosees())
               return boundarycase;
            else
               return "no boundary";
         }
      }


   //more vs less
      public String comparison (String distinguisher, Case cfs, Case citedcase)
      {
         if (cfs.hasDimension(this) && citedcase.hasDimension(this))
         {
            if (distinguisher.equals("plaintiff"))
            {                                        //since it favors plaintiff
               if (cfs.numDisclosees() < citedcase.numDisclosees())
               {
                  return "In the cited case, " + citedcase.caseName() + ", " + magnitude(c
                     + ", whereas in the current fact situation " + magnitude(cfs);
               }
               else
                  return "not distinguished";
            }
            else
```

```
        {
            if (cfs.numDisclosees() < citedcase.numDisclosees())
                return "not distinguished";
            else
            {
                return "In the current fact situation, " + magnitude(cfs)
                    + ", whereas in the cited case, " + citedcase.caseName() + ", " + mag
            }
        }
    }
    else if (cfs.hasDimension(this) && !(citedcase.hasDimension(this)))
    {
        if (favors.equals(distinguisher))
            return "In the current fact situation, " + cfs.caseName() + ", "
                + description + ".  Not so in the cited case, " + citedcase.caseName() +
        else
            return "not distinguished";
    }
    else
    {
        if (favors.equals(distinguisher))
            return "not distinguished";
        else
            return "In the cited case, " + citedcase.caseName() + ", "
                + description + ".  Not so in the current fact situation, " + cfs.caseNa
    }
  }
}
```

```
public class NondisclosureAgreementSpecific extends Dimension
{
   private String name = "Nondisclosure Agreement Specific";
   private String favors = "plaintiff";
   private String description = "THE NONDISCLOSURE AGREEMENT REFERRED SPECIFICALLY TO T
   private String comparisontype = "some vs none";
   private String proplaintdirection = "some";

   public NondisclosureAgreementSpecific()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return description;
   }
   public String description()
   {
      return description;
   }
   public String magnitude(Case thecase)
   {
      return "The employee , " + thecase.commonEmployeeName() + ", was bound by a "
         + "nondisclosure agreement with his/her original employer, the plaintiff, "
         + thecase.plaintiff() + " which referred specifically to the plaintiff's produ
         + thecase.plaintiffProduct();
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.partiesCompete() == false)
         return "not applicable";
      if (thecase.disclosureEvent() == false)
         return "not applicable";
      if (thecase.nondisclosureAgreement().equals("no"))
         return "not applicable";
      if (thecase.nondisclosureAgreement().equals("yes"))
         return "near miss";
      else
```

```
        return "applicable";
    }

    public String comparison (String distinguisher, Case cfs, Case citedcase)
    {
       // this comparison method is good for any dimension where the
       // comparison type is some vs. none.  The magnitude could
       // be included in the distinguishing comments too
       if (cfs.hasDimension(this) && citedcase.hasDimension(this))
       {
          return "not distinguished";
       }
       else if (cfs.hasDimension(this) && !(citedcase.hasDimension(this)))
       {
          if (favors.equals(distinguisher))
             return "In the current fact situation, " + cfs.caseName() + ", "
                + description + ".  Not so in the cited case, " + citedcase.caseName() +
          else
             return "not distinguished";
       }
       else
       {
          if (favors.equals(distinguisher))
             return "not distinguished";
          else
             return "In the cited case, " + citedcase.caseName() + ", "
                + description + ".  Not so in the current fact situation, " + cfs.caseNa
       }
    }
}
```

```
public class NondisclosureAgreement extends Dimension
{
   private String name = "Nondisclosure Agreement";
   private String favors = "plaintiff";
   private String description = "EMPLOYEE ENTERED INTO A NONDISCLOSURE AGREEMENT WITH T
   private String comparisontype = "some vs none";
   private String proplaintdirection = "some";

   public NondisclosureAgreement()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return description;
   }
   public String description()
   {
      return description;
   }
   public String magnitude(Case thecase)
   {
      return "The employee , " + thecase.commonEmployeeName() + ", was bound by a "
         + "nondisclosure agreement with his/her original employer, the plaintiff, "
         + thecase.plaintiff();
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.partiesCompete() == false)
         return "not applicable";
      if (thecase.disclosureEvent() == false)
         return "not applicable";
      if (thecase.nondisclosureAgreement().equals("no"))
         return "near miss";
      else
         return "applicable";
   }
```

```
   public String comparison (String distinguisher, Case cfs, Case citedcase)
   {
      // this comparison method is good for any dimension where the
      // comparison type is some vs. none.  The magnitude could
      // be included in the distinguishing comments too
      if (cfs.hasDimension(this) && citedcase.hasDimension(this))
      {
         return "not distinguished";
      }
      else if (cfs.hasDimension(this) && !(citedcase.hasDimension(this)))
      {
         if (favors.equals(distinguisher))
            return "In the current fact situation, " + cfs.caseName() + ", "
               + description + ".  Not so in the cited case, " + citedcase.caseName() +
         else
            return "not distinguished";
      }
      else
      {
         if (favors.equals(distinguisher))
            return "not distinguished";
         else
            return "In the cited case, " + citedcase.caseName() + ", "
               + description + ".  Not so in the current fact situation, " + cfs.caseNa
      }
   }
}
```

```
public class NoncompetitionCovenant extends Dimension
{
   private String name = "Noncompetition Covenant";
   private String favors = "plaintiff";
   private String description = "EMPLOYEE ENTERED INTO A NONCOMPETITION COVENANT WITH T
   private String comparisontype = "some vs none";
   private String proplaintdirection = "some";

   public NoncompetitionCovenant()
   {
   }
   public String toString()
   {
      return name;
   }
   public String name()
   {
      return name;
   }
   public String favors()
   {
      return favors;
   }
   public String favorDescription()
   {
      return description;
   }
   public String description()
   {
      return description;
   }
   public String magnitude(Case thecase)
   {
      return "The employee , " + thecase.commonEmployeeName() + ", was bound by a "
         + "noncompetition covenant with his/her original employer, the plaintiff, "
         + thecase.plaintiff();
   }
   public String test (Case thecase)
   {
      if (thecase.plaintiff() == null)
         return "not applicable";
      if (thecase.plaintiffProduct() == null)
         return "not applicable";
      if (thecase.defendant() == null)
         return "not applicable";
      if (thecase.defendantProduct() == null)
         return "not applicable";
      if (thecase.productsCompete() == false)
         return "not applicable";
      if (thecase.commonEmployeeName() == null)
         return "not applicable";
      if (thecase.partiesCompete() == false)
         return "not applicable";
      if (!(thecase.noncompetitionCovenant()))
         return "near miss";
      else
         return "applicable";
```

```
    }

    public String comparison (String distinguisher, Case cfs, Case citedcase)
    {
       // this comparison method is good for any dimension where the
       // comparison type is some vs. none.  The magnitude could
       // be included in the distinguishing comments too
       if (cfs.hasDimension(this) && citedcase.hasDimension(this))
       {
          return "not distinguished";
       }
       else if (cfs.hasDimension(this) && !(citedcase.hasDimension(this)))
       {
          if (favors.equals(distinguisher))
             return "In the current fact situation, " + cfs.caseName() + ", "
                + description + ".  Not so in the cited case, " + citedcase.caseName() +
          else
             return "not distinguished";
       }
       else
       {
          if (favors.equals(distinguisher))
             return "not distinguished";
          else
             return "In the cited case, " + citedcase.caseName() + ", "
                + description + ".  Not so in the current fact situation, " + cfs.caseNa
       }
    }
}
```